UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RÉGES EDUARDO OBERDERFER JÚNIOR

Deep Learning for Boolean Matching

Trabalho de conclusão de graduação.
Trabalho realizado na UFRGS dentro do acordo de dupla diplomação UFRGS – EMA.

Orientador: Prof. Dr. André Inácio Reis

Porto Alegre
Julho de 2018

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Oberderfer Júnior, Réges Eduardo

Deep Learning for Boolean Matching / Réges Eduardo Oberderfer Júnior. – Porto Alegre: Engenharia de Computação da UFRGS, 2018.

59 f.: il.

Trabalho de conclusão (Bacharelado) – Universidade Federal do Rio Grande do Sul. Bacharelado em Engenharia de Computação, Porto Alegre, BR-RS, 2018. Orientador: Prof. Dr. André Inácio Reis.

1. Boolean Matching. 2. Deep Learning. 3. Mapeamento Tecnológico. 4. Síntese Lógica. I. Reis, André Inácio. II. Título.

RESUMO

Boolean matching é a tarefa de determinar a equivalência entre funções booleanas, uma etapa essencial no mapeamento tecnológico. Este trabalho explora um novo método para resolver boolean matching usando aprendizagem de máquina. Boolean matching é aplicado usando um jogo de aprendizagem por reforço combinado com redes neurais profundas. O objetivo do jogo é encontrar a função representante semi-canônica, que é tomada como a função que corresponde ao menor inteiro na classe de equivalência. Se para duas funções a mesma função resultante é obtida, elas são equivalentes. Resultados mostram que o método proposto teve pior qualidade de resultados e pior desempenho do que o método usado para comparação. Porém, existe espaço para melhorar a performance usando estruturas de dados mais eficientes e hardware específico.

Palavras-chave: Boolean Matching, Deep Learning, Mapeamento Tecnológico, Síntese Lógica.

Resumo estendido

Este é um resumo estendido em português para a Universidade Federal do Rio Grande do Sul do trabalho original que segue. O trabalho de conclusão original, em inglês, foi apresentado na UFRGS através do programa de dupla diplomação entre a UFRGS e a École des Mines d'Alès.

1. Introdução

Determinar se duas funções booleanas são equivalentes independentemente de permutação e negação de entradas e de negação da saída (NPN) é um problema muito importante na área da síntese lógica. A síntese lógica é uma das etapas no fluxo standard cell em VLSI. O mapeamento tecnológico, na fase dependente de tecnologia da síntese lógica, consiste em buscar portas lógicas definidas numa biblioteca para implementar funções booleanas num circuito [Mailhot e Micheli 1990]. Essa busca deve ser capaz de encontrar a equivalência entre funções NPN. Duas funções f1 e f2 são NPN-equivalentes se é possível transformar f1 em f2 usando permutação e/ou negação de entradas e/ou negação da saída. Boolean matching é a tarefa de determinar a equivalência entre funções booleanas, uma tarefa essencial no mapeamento tecnológico [Kapoor 1995].

Existem diversos métodos para realizar Boolean matching. Alguns deles se baseiam em formas canônicas de representação de modo que, se ambas as funções f1 e f2 resultam num mesmo representante canônico, elas são equivalentes [Abdollahi e Pedram 2008][Petkovska et al. 2016][Zhang et al. 2017a]. Outros métodos buscam a relação de correspondência entre as variáveis das duas funções. Dessa forma, esses métodos encontram a transformação que deve ser aplicada a uma das funções para que a outra seja obtida [Abdollahi 2008][Zhang et al. 2017b]. Existem ainda algoritmos baseados em SAT, que transformam o problema de Boolean matching em uma fórmula booleana [Katebi e Markov 2010].

1.1. Motivação

Nos últimos anos, aprendizagem de máquina, que era uma simples ferramenta para reconhecimento de padrões, tornou-se capaz de vencer um humano em um jogo de Go [Silver et al. 2016]. Várias tarefas como controle de membros robóticos [Mnih et al. 2015] são realizadas através da aprendizagem por reforço que foi revolucionada por avanços recentes em deep learning. O sucesso desse método está na capacidade de encontrar automaticamente características especiais e construir modelos hierárquicos para um dado problema. Isso só é possível devido ao uso combinado de aprendizagem por reforço e redes neurais.

Empresas que trabalham com EDA vêm investindo em aprendizagem de máquina [Bailey 2017]. O seu interesse é tomar proveito desse método automático para acelerar o processo de convergência do fluxo de projeto de circuitos integrados. Devido à importância do Boolean matching na fase de mapeamento tecnológico e ao interesse das empresas de EDA, este trabalho explora um novo método para resolver Boolean matching usando aprendizagem de máquina.

1.2. Objetivo

O objetivo deste trabalho é expandir o estudo da aplicação de métodos de aprendizagem de máquina em síntese lógica, iniciado por [Haaswijk et al. 2017], que usa deep learning para otimização lógica. Note que o trabalho proposto em [Haaswijk et al. 2017] é pioneiro no uso de aprendizagem de máquina para síntese lógica, e esta área ainda está começando a ser explorada. Assim acreditamos que nosso trabalho será pioneiro na área

matching de células de biblioteca usando técnicas de aprendizagem de máquina, e a experiência adquirida durante o trabalho será importante para o entendimento do potencial de aplicação de aprendizagem de máquina em outros problemas de síntese lógica.

2. Método Proposto

Este trabalho propõe resolver o problema de Boolean matching NPN com um jogo de aprendizado por reforço inspirado nos métodos baseados em formas canônicas. As funções booleanas são representadas por inteiros extraídos da última coluna da tabela verdade. Cada função corresponde a um estado do jogo. As ações do jogo são o conjunto de todas as possíveis transformações NPN. O objetivo do jogo é encontrar uma sequência finita de transformações NPN a serem aplicadas a uma função de entrada. A aplicação dessas transformações à função de entrada deve obter uma função de saída que é a representate semi-canônica. A tomada de decisão sobre as transformações escolhidas no jogo é feita por uma rede neural profunda.

3. Resultados

Os método foi comparado ao trabalho de [HUANG et al. 2013] usando uma métrica de desempenho versus qualidade dos resultados. Os resultados mostram que o método proposto teve valores superior em número de classes de equivalência, porém o tempo de execução foi maior. Além disso, as execuções para funções parcialmente DSD e não-DSD tomaram mais tempo do que aquelas para funções DSD. Sobre o método proposto, apesar de que o tempo de execução até 12 variáveis variou para os diferentes benchmarks, o tempo para 14 variáveis foi bem similar em todos. Isso pode ser explicado pelo fato de que, à medida que o número de variáveis aumenta, também aumenta o tamanho da tabela verdade. Consequentemente, o tamanho da entrada da rede neural cresce. Quanto maior o tamanho da entrada, mais neurônios têm de ser usados para escolher a ação a ser executada, até um ponto em que quase a rede inteira é ativada simultaneamente. Isso poderia ser melhor analizado se os benchmarks fossem executados para 16 variáveis. Entretando, não havia memória suficiente no sistema de 8 GB usado para os testes.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RÉGES EDUARDO OBERDERFER JÚNIOR

Deep Learning for Boolean Matching

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Engeneering

Advisor: Prof. Dr. André Inácio Reis

Porto Alegre
June 2018

“The more sand has escaped from the hourglass of our life,
the clearer we should see through it.”
— NICCOLÒ MACHIAVELLI

ACKNOWLEDGEMENTS

I would like to dedicate this work to my parents Réges and Leia and to my girlfriend Amanda, who have been of utmost importance in my final years of graduation.

I am very grateful to my advisor, Professor André Inácio Reis, for his advice and technical aid. I am also grateful to Professor Renato Perez Ribas for his guidance and advice on decision making.

Special thanks to Vinicius Possani and to everyone in the Logic Circuit Synthesis (LogiCS) lab for all the time they have dedicated to discussing and giving constructive criticism to my ideas.

I also thank the members of this work's committee, Professor Sérgio Bampi and Professor Fernanda Gusmão de Lima Kastensmidt, for their comments and for helping to improve this work.

ABSTRACT

Boolean matching is the task of determining equivalence between boolean functions, a key step in technology mapping. This work proposes a new method to solve boolean matching using deep learning. A reinforcement learning approach combined with a deep neural network is used to match boolean functions recasting matching as a game. The game's objective is to find a semi-canonical representative function, which is a function that corresponds to the smallest integer in the equivalence class. If the output of the method is the same for two functions, they are equivalent. Results show that the proposed method has worse quality of results and performance than the compared method. However, there is space for improving performance by using more efficient data structures and specific hardware.

Keywords: Boolean Matching. Deep Learning. Technology Mapping. Logic Synthesis.

Deep Learning para Equivalência de Funções Booleanas

RESUMO

Boolean matching é a tarefa de determinar a equivalência entre funções booleanas, uma etapa essencial no mapeamento tecnológico. Este trabalho explora um novo método para resolver boolean matching usando aprendizagem de máquina. Boolean matching é aplicado usando um jogo de aprendizagem por reforço combinado com redes neurais profundas. O objetivo do jogo é encontrar a função representante semi-canônica, que é tomada como a função que corresponde ao menor inteiro na classe de equivalência. Se para duas funções a mesma função resultante é obtida, elas são equivalentes. Resultados mostram que o método proposto teve pior qualidade de resultados e pior desempenho do que o método usado para comparação. Porém, existe espaço para melhorar a performance usando estruturas de dados mais eficientes e hardware específico.

Palavras-chave: Boolean Matching, Deep Learning, Mapeamento Tecnológico, Síntese Lógica.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter Graph
BDD	Binary Decision Diagram
DSD	Disjoint-Support Decomposable
EDA	Electronic Design Automation
MIG	Majority-Inverter Graph
TPU	Tensor Processing Unit
VLSI	Very Large Scale Integration

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Determining if two boolean functions are equivalent regardless of negation of input variables (flips), permutation of input variables (swaps) and negation of the output is a key problem in logic synthesis. Logic synthesis is one of the main steps in a standard cell flow for VLSI. Technology mapping, in the technology dependent step of logic synthesis, searches for logic gates defined in a library in order to implement boolean functions in a logic circuit (MAILHOT; MICHELI, 1990). This search must be able to find the NPN-equivalence of boolean functions. Given two functions f_1 and f_2, f_1 is NPN-equivalent to f_2 if it is possible to transform f_1 into f_2 by only applying flips, swaps and output negation. Boolean matching is the task of determining the equivalence of boolean functions, an important task in technology mapping (KAPOOR, 1995).

There are several methods for solving boolean matching. Some of them are based on canonical forms of representation so that if two functions f_1 and f_2 are equivalent if they can be matched with the same canonical representant (ABDOLLAHI; PEDRAM, 2008)(PETKOVSKA et al., 2016)(ZHANG et al., 2017a). Other methods look for correspondences in the variable supports of the functions to find the transformation which must be applied to one of the functions for obtaining the other one (ABDOLLAHI, 2008)(ZHANG et al., 2017b). There are also SAT-based methods, which transform the boolean matching problem in a boolean formula (KATEBI; MARKOV, 2010).

The authors Petkovska et al. (2016) use a hierarchical method to determine NPN equivalence of boolean functions. The hierarchical method keeps intermediate information that is reused later, increasing performance. Intermediate equivalence classes are generated and the final equivalence classes and their transformations are obtained by transforming the input function in its intermediate representative and then into its final representative.

Zhang et al. (2017a) present an algorithm that uses boolean difference and cofactor. The signature vector and symmetry properties of variables are used to match boolean functions with canonical forms. The symmetry properties reduce the search space in order to obtain superior performance in comparison with previous methods.

Abdollahi (2008) present for the first time a signature based method to determine equivalence of incompletely specified boolean functions. The method uses BDDs to represent the boolean functions, one BDD for its ON-set and another for its OFF-set. Signatures extracted from the BDDs are used to create a bipartite graph. The graph is reduced

and an algorithm of branch and bound is applied to find the matching transformations.

Zhang et al. (2017b) propose a structural signature vector to determine boolean function equivalence. Symmetry properties are used to reduce search space. The algorithm is based on a recursive decomposition and search strategy.

Katebi and Markov (2010) show a compound approach that combines AIGs, simulation and SAT to determine boolean equivalence. AIGs are chosen over BDDs because the method works for large scale circuits. If neither AIGs nor simulation are enough to solve an instance, SAT is applied.

1.1 Motivation

Machine learning, which used to be only a tool for recognizing patterns, has recently been able to beat a human player in the Go game (SILVER et al., 2016). Several tasks, such as fine controling robot members, are now done by reinforcement learning techniques which have benefited greatly from advancements in *deep learning*. Due to the combination of reinforcement learning and deep neural networks, new solutions can be found for certain problems.

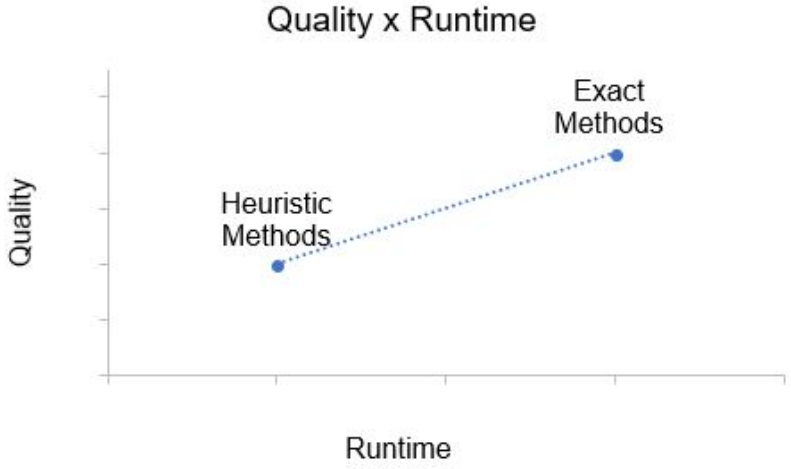
Companies that work with EDA have been investing in machine learning (BAILEY, 2017). They look forward to taking advantage of and automated method to reduce the time taken to synthesize a circuit in a design flow. Given the importance of boolean matching in technology mapping and to the interest of EDA companies, this work explores a new method to solve boolean matching using reinforcement learning and deep neural networks.

The main goal of the investing companies is to benefit from the trade-off between quality of results and runtime, shown in the trend line in figure 1.1, that machine learning as a heuristic can establish. Since exact methods may be too computationally expensive, there is a need for heuristics that can be fulfilled by machine learning approaches.

1.2 Objective

The objective of this work is to expand the study of the application of machine learning methods in logic synthesis, started by Haaswijk, Collins and Seguin (2017) . The work proposed in (HAASWIJK; COLLINS; SEGUIN, 2017) applies deep learning

Figure 1.1: Quality x Runtime for exact and heuristic methods.



in logic optimization and is the first to use machine learning in logic synthesis. Since this area has been explored very little, we believe that this work will be a pioneer in applying machine learning for boolean matching. Therefore, the experience acquired during this work will be important to understand the potential of using machine learning in other logic synthesis problems.

1.3 Structure of the Text

This text is organized as follows. Chapters 2 and 3 present the background on boolean functions and machine learning, respectively. Chapter 4 describes the proposed method. Chapter 5 shows the results and discussions. Chapter 6 concludes and talks about future work.

2 BOOLEAN MATCHING

2.1 Boolean Functions

Given $B = \{0, 1\}$ and $O = B \cup \{\mathbf{X}\}$, where X is the symbol that represents *don't care*, an n-input *boolean function* f is a mapping

$$f : B^n \rightarrow O \tag{2.1}$$

A *completely specified boolean function* is a mapping

$$f : B^n \rightarrow B \tag{2.2}$$

that only outputs $\{0, 1\}$.

2.1.1 Representing Boolean Functions

One of the most common representations for boolean functions is the *truth table*, which lists all the outputs for all possible combination of input values. An example of truth table is shown in table 2.1.

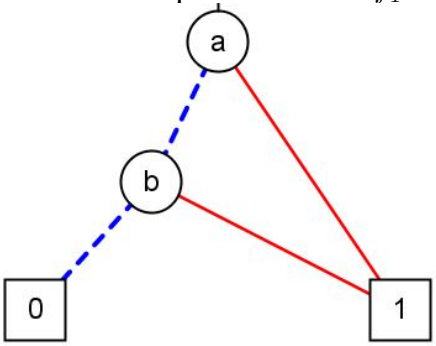
Table 2.1: Truth table representation example for $f_1 = a + b$.

a	**b**	**f**
0	0	0
0	1	1
1	0	1
1	1	1

Given that the variable order of a boolean function is known, the function may also be represented by the last column of its truth table in the form of an integer. **Example:** The variable order of f_1 as seen in table 2.1 is $[a, b]$. Thus, $f_1 = a + b = 0111_2 = 7_{16}$.

Another way to represent boolean functions is with Binary Decision Diagrams (BDD). BDDs are rooted, directed graphs with two types of nodes. The *terminal* nodes have a value B. The *nonterminal* nodes behave like an *if then else*: they have an input and two outputs, one in case the input's value is equal to 1 and another if it is equal to 0. Figure 2.1.1 shows the BDD representation of $f_1 = a + b$, where the 1 outputs are depicted in blue and the 0 outputs, in red.

Figure 2.1: BDD representation of $f_1 = a + b$.



2.2 Disjoint-support Decomposable Functions

A boolean function can be functionally decomposed by two subfunctions, g and h, called composition and decomposition functions respectively. Equation 2.3 shows a functional decomposition of f, where $X = X_1 \cup X_2$.

$$f(X) = h(g(X_1), X_2) \tag{2.3}$$

If X_1 and X_2 do not share any elements, the functional decomposition is called a disjoint-support decomposition (DSD) and f is disjoint-support decomposable.

2.3 Cofactor of a Boolean Function

Given a boolean function f with inputs $[e_1, e_2, ..., e_n]$, the *cofactor* of f for an input variable e_i is the evaluation of e_i in function f (BOOLE, 1853).

$$f_{e_i} = f(e_i = 1) \tag{2.4}$$

$$f_{\overline{e_i}} = f(e_i = 0) \tag{2.5}$$

The *positive* cofactor of f for the variable e_i is f_{e_i}, as shown in equation 2.4. The *negative* cofactor of f for the variable e_i is $f_{\overline{e_i}}$, as shown in equation 2.5.

2.4 Unateness and Symmetry of Variables

The *unate* and *binate* properties are important to understand the behavior of variables in a boolean function. Given the positive and negative cofactors of f for input variables e_i and e_j, the behavior of e_i in function f is as shown in table 2.2.

Table 2.2: Behavior of variables of a boolean function.

Relation	**Behavior**
$f_{e_i} \equiv f_{\overline{e_i}}$	$don't\ care$
$(f_{e_i} \neq f_{\overline{e_i}}) \wedge (f_{e_i} \equiv f_{e_i} + f_{\overline{e_i}})$	$positive\ unate$
$(f_{e_i} \neq f_{\overline{e_i}}) \wedge (f_{\overline{e_i}} \equiv f_{e_i} + f_{\overline{e_i}})$	$negative\ unate$
$(f_{e_i} \neq f_{\overline{e_i}}) \wedge (f_{e_i} \neq f_{e_i} + f_{\overline{e_i}}) \wedge (f_{\overline{e_i}} \neq f_{e_i} + f_{\overline{e_i}})$	$binate$
$f_{e_i,\overline{e_j}} \equiv f_{\overline{e_i},e_j}$	e_i is $symmetric$ to e_j

2.5 Equivalence Classes of Boolean Functions

Permuting two input variables is called a $swap$ ($ab \rightarrow ba$). Negating an input variable is called a $flip$ ($a \rightarrow \overline{a}$). It is also possible to negate the output. These three operations are the *transformations* that can be applied to a boolean function (HUANG et al., 2013). Two boolean functions belong to the same *equivalence class* if it is possible to apply a series of transformations to one of the functions so that the second is obtained. Table 2.3 shows the taxonomy of equivalence classes.

Table 2.3: Equivalence classes of boolean functions.

Transformations	**Equivalence class**
Swaps	P
Flips and swaps	NP
Swaps and output negation	PN
Flips, swaps and output negation	NPN

If two boolean functions belong to the same equivalence class P, NP, PN or NPN, they are P, NP, PN or NPN-equivalent, respectively.

Example: Given the functions $f_1 = a * b + c$, $f_2 = a + b * c$ e $f_3 = \overline{a} + b * c$, $f_4 = \overline{a + b * c}$ e $f_5 = \overline{\overline{a} + b * c}$. Table 2.4 shows the equivalence relations between them.

Table 2.4: Equivalence relations between boolean functions.

Functions	**Equivalence**	**Transformations**
f_1 e f_2	P, NP, PN and NPN	swap(a,c)
f_1 e f_3	NP and NPN	swap(a,c) and flip(a)
f_1 e f_4	PN and NPN	swap(a,c) and negate the output
f_1 e f_4	NPN	swap(a,c), flip(a) and negate the output

2.6 Boolean Matching

Determining if two boolean functions are equivalent is a problem called *boolean matching*. In order for two boolean functions to be equivalent, they must belong to the same equivalence class. Thus, P-matching, NP-matching, PN-matching and NPN-matching are instances of the boolean matching problem of each of the equivalence classes respectively.

Let $f(X)$ and $g(Y)$ be two boolean functions, where $X = \{x_1, x_2, ..., x_n\}$ and $Y = \{y_1, y_2, ..., y_n\}$. A *match* is a variable mapping so that $m(X) : Y$ and $f(X) = g(m(X))$.

Example: Let $f_1(X) = \overline{x_1}x_2 + x_3\overline{x_4}$, $g_1(X) = y_1\overline{y_2} + \overline{y_3}y_4$ and $f_1(X) = g_1(m_1(X))$. A possible variable mapping m_1 is given by equation 2.6.

$$\begin{aligned} x_1 &\rightarrow y_2 \\ x_2 &\rightarrow y_1 \\ x_3 &\rightarrow y_4 \\ x_4 &\rightarrow y_3 \end{aligned} \tag{2.6}$$

2.7 Canonical Forms

A *canonical form of representation* is any representation that is unique, i.e there can be no more than one representant for a given element, object or class. Thus, a canonical form for equivalence classes of boolean functions is a representation that defines one and exactly one representant for each class. Often the representant, also called *signature*, is a function with special properties that is chosen among the set of functions which belong to the equivalence class.

A widely used canonical form is given by selecting as signature the function whose truth table integer is the smallest. Since each truth table corresponds to exactly one function, this representation is unique. However, previous works have shown that finding

exactly the smallest integer truth table can be computationally expensive (HUANG et al., 2013). Therefore, there exists a need for faster canonical forms.

2.7.1 Semi-canonical Forms

Semi-canonical forms are also unique forms of representation, but they are more relaxed than canonical forms. Even though a semi-canonical form may also have as goal to find the smallest integer truth table, it does so with certain constraints on its search space. For example, a semi-canonical form may not explore all the possible variable swaps, therefore the semi-canonical representant might not be the smallest integer truth table, but the smallest that can be found according to the swaps that are considered.

The semi-canonical form presented in (YANG; WANG; MISHCHENKO, 2012) only considers swaps of adjacent input variables. According to Yang, Wang and Mishchenko (2012), the number of semi-canonical classes obtained is not prohibitively large. Due to the fact that a lesser number of swaps is considered, this semi-canonical form is significantly faster than that of canonical forms.

2.8 The ABC Tool

The ABC, which stands for A System for Sequential Analysis and Verification, is a tool from Berkeley Logic Synthesis and Verification Verification Group maintained by Professor Alan Mishchenko. The ABC includes several features, such as data structures for representing netlists, logic networks, BDDs and AIGs, parsers for BLIF, PLA and BENCH formats, efficient combinational synthesis flow algorithms based on AIGs, technology mappers for standard cells and FPGA, commands for LUT mapping and SAT-based combinational equivalence checking.

One of the ABC commands for technology mapping is the *testnpn* command, which computes semi-canonical forms for completely specified boolean functions of up to 16 variables. It takes as input a file of truth tables in hexadecimal format and outputs their semi-canonical forms as proposed by Yang, Wang and Mishchenko (2012). The *testnpn* command usage is shown in figure 2.2.

Figure 2.2: testnpn command usage.

```
abc 01> testnpn -h
usage: testnpn [-AN <num>] [-dbvh] <file>
                   testbench for computing (semi-)canonical forms
                   of completely-specified Boolean functions up to 16 varibles
        -A <num> : semi-caninical form computation algorithm [default = 0]
                       0: uniqifying truth tables
                       1: exact NPN canonical form by brute-force enumeration
                       2: semi-canonical form by counting 1s in cofactors
                       3: Jake's hybrid semi-canonical form (fast)
                       4: Jake's hybrid semi-canonical form (high-effort)
                       5: new fast hybrid semi-canonical form
                       6: new phase canonical form
                       7: new hierarchical matching
                       8: hierarchical matching by XueGong Zhou at Fudan University, Shanghai
        -N <num> : the number of support variables (binary files only) [default = unused]
        -d       : toggle dumping resulting functions into a file [default = no]
        -b       : toggle dumping in binary format [default = no]
        -v       : toggle verbose printout [default = no]
        -h       : print the command usage
        <file>   : a text file with truth tables in hexadecimal, listed one per line,
                   or a binary file with an array of truth tables (in this case,
                   -N <num> is required to determine how many functions are stored)
```

3 MACHINE LEARNING

Machine learning can be divided in three main branches: *supervised*, *unsupervised*, and *reinforcement* learning. This work focuses mainly on reinforcement learning and secondly on supervised learning.

- *Supervised* learning is applied on data whose inputs and outputs are known, also called structured data. The model learns from the known information in order to be later applied to new data whose outputs are unknown.
- *Unsupervised* learning tries to find patterns on unstructured data, either by clustering or association.
- *Reinforcement* learning attempts to maximize its reward and learns by trial and error.

3.1 Artificial Neurons

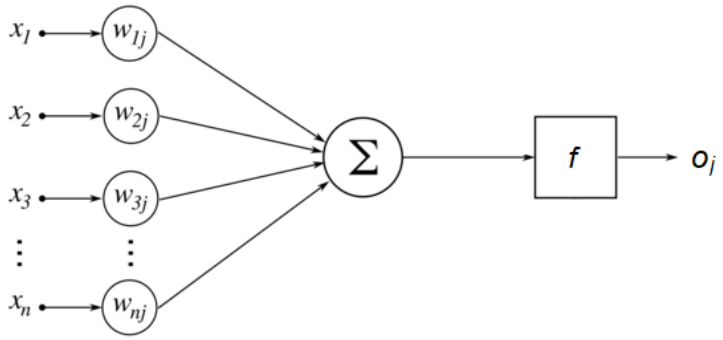
Artificial neurons are a mathematical model of biological neurons and also a part of the supervised machine learning methods. Their main function is to output a number which is a function of the neuron's input values and input weights.

The notation for artificial neurons used in this work is shown in table 3.1. Figure 3.1 highlights an artificial neuron's inputs $X = [x_1, x_2, x_3, ..., x_n]$, weights $W = [w_{1j}, w_{2j}, w_{3j}..., w_{nj}]$, its activation function f and its output o_i.

Table 3.1: Notation for artificial neurons.

Symbol	Description
n	number of neurons of current layer
i	neuron i of a layer
j	neuron j of the layer next to they layer of i
w_{ij}	weight of the edge that connects neuron i to neuron j
o_j	output of neuron j
v_j	input of neuron j
d_j	expected output of neuron j
e_j	error derivative of neuron j
f	activation function
f'	activation function derivative
α	learning rate

Figure 3.1: Model of an artificial neuron. (WANG, 1995)



3.2 Activation Functions

An activation function is a function that takes as input the sum of the input values of a neuron multiplied by their weights, as shown in equation 3.1, and outputs a value in interval $[0, 1]$ or $[-1, 1]$. Activation functions are a key part of an artificial neuron, because they provide neurons with a non-linear behavior that allows them to approximate non-linear patterns (LAU; LIM, 2017). Without these functions, an arrangement of several artificial neurons in a network would only be able to produce outputs that are linear combinations of the inputs. Equations 3.2, 3.3, 3.4 and 3.5 show, respectively, the activation functions *Heaviside*, *ReLU*, *Logistic* and *Tanh*.

$$o_j = f(\sum_{i=1}^{n} w_{ij}\, o_i) \tag{3.1}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{3.2}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{3.3}$$

$$f(x) = \frac{1}{1 + \exp^{-x}} \tag{3.4}$$

$$f(x) = \frac{2}{1 + \exp^{-2x}} - 1 \tag{3.5}$$

3.3 Output Error Derivative

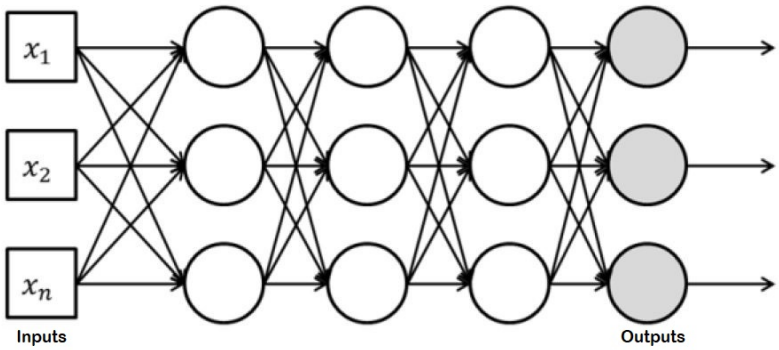
The output error of a neuron can be simply calculated by evaluating the difference between its output value and its expected output value. The output error derivative is shown in equation 3.6.

$$e_i = f'(o_i)(d_i - o_i) \tag{3.6}$$

3.4 Neural Networks

A neural network is a statistical model of a real world system made by several layers of neurons. It is also a supervised machine learning technique. The network uses parameters known as *weights* to propagate information from a neuron to the neurons of the next layer in order to detect patterns. Data is inserted in the *input layer* and advances through *hidden layers* until the *output layer* is reached. That is, the neural network maps input values to output values. The weights are adjusted to correctly map each input value to its corresponding output value in a dataset. The adjustment of the weights is done in the *training* step by a process called backpropagation, which propagates the output error derivative and uses each neuron's contribution to the output error to better adjust the weights as training progresses. Figure 3.4 shows a simple neural network.

Figure 3.2: Model of a simple neural network. (ASSODIKY; SYARIF; BADRIYAH, 2017)



3.4.1 Training a Neural Network

In order to train a neural network, a training dataset is needed. A training dataset is a set of observations with numerical or categorical values for several *attributes* (attr.), as seen in table 3.2. Some of these attributes, the *target attributes*, are known during the training phase but unknown during the testing phase. The target attributes must be predicted by the neural networks during the testing phase.

Table 3.2: A training dataset.

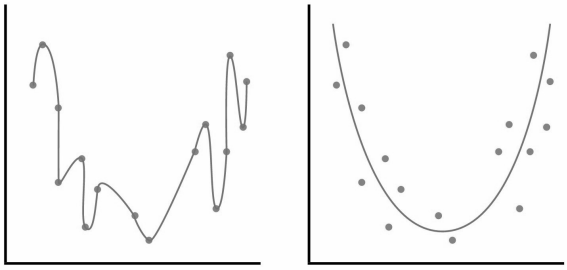
Observation	**Attr. 1**	**Attr. 2**	**Attr. 3**	**Target Attr. 1**	**Target Attr. 2**
x_1	3	0.1	4	1	0.5
x_2	7	0.3	6	0	1
x_3	4	0.2	7	1	0

The training of a neural network is an iterative process of inserting data from the attributes into the input layer and observing the output error. At each iteration or at each batch of observations, the networks' weights are adjusted to the data. The goal of the training phase is to reduce the output error of the network according to the training dataset.

3.4.2 Overfitting

After training the neural network, it is necessary to validate it with a testing dataset. The testing dataset is a small part of the data - in general, 10% to 20% - which is put aside from all the available when the training dataset is selected. The goal of the testing phase is to verify if the network is overly adjusted to the training dataset. Figure 3.3 shows on the left an overly adjusted model, this is known as *overfitting*. An overfitted model has a big error for the testing dataset when compared to its error for the training dataset because the pattern recognized by the model is more complex than that of the actual data. A solution to this problem is to compare both of these errors and simplify the model if overfitting occurs. In a neural network, the model can be simplified by reducing the number of neurons or the number of layers.

Figure 3.3: To the left, a model overly adjusted to the data (overfitting). To the right, a well adjusted model. (KAGGLE, 2017)



3.4.3 Backpropagation Algorithm

In a neural network, the output error derivative may be propagated from the outputs to the inputs by the *backpropagation* algorithm. Initially, the error derivative for the output layer is calculated using equation 3.6. Then, these error derivatives are propagated back using equation 3.7. Finally, the weights are updated using equations 3.8 and 3.9.

$$e_i = f'(o_i) \sum_{j=1}^{n} e_j \, w_{ij} \tag{3.7}$$

$$\Delta w_{ij} = \alpha \, e_j \, o_i \tag{3.8}$$

$$new\_w_{ij} = w_{ij} + \Delta w_{ij} \tag{3.9}$$

The backpropagation is given by the following algorithm (SWINGLER, 1996).

1. Initialize all weights with random values in $[0, 1]$ or $[-1, 1]$
2. **Repeat**
    1. Choose an observation from the training dataset.
    2. Copy the attribute values to the input layer.
    3. Propagate the input through all the hidden layers using the weights and the activation function.
    4. Calculate the error derivative for the output layer.
    5. Propagate back the output error derivative.
    6. Update the weights.
7. **Until** the output error is sufficiently small.

3.5 Reinforcement Learning

Reinforcement learning maximizes the long-term reward by trial and error. In order to to that, reinforcement learning searches for the optimal policy π^* which maps each state s to the probability of each action a be chosen with the goal of maximizing the reward r in the long-term (NGUYEN; NGUYEN; NAHAVANDI, 2017). The mapping function Γ_π is given by equation 3.10, where $\Lambda_\pi(s)$ is the space of actions from state s following policy π.

$$\Gamma_\pi = \{P(s \xrightarrow{a} s'|\pi) : \forall a \in \Lambda_\pi(s)\} \tag{3.10}$$

3.5.1 Policies

Policies are the mechanisms used to select the action to be performed on a given state. There are several ways of choosing the next action:

- The *Boltzmann policy* builds a Boltzmann probability law on the probabilities of choosing each action and selects an action randomly according to this law.
- The *Greedy policy* always selects the best action according to the model.
- The *Epsilon Greedy policy* either selects the best action according to the model with probability $1 - epsilon$ or selects a random action with probability $epsilon$.

Different policies can be applied for the training and testing phases. It is important that the policy selected for the training phase sometimes choose actions randomly in order to explore different paths and determine which one is the best. If no random choices are made, the model may lock itself onto a local optimum.

3.6 The Keras API

Keras is a high-level modular neural network API available for Python. It is designed to work with TensorFlow, Theano or CNTK and allows users to quickly experiment with different models of neural networks. Keras provides an API that enables the user to combine different neural networks modules as sequential layers. Keras layers include *input* layers; *dense* layers of neurons; *activation* layers, which implement activation

functions; *dropout* layers, which randomly drops a part of the weights at each training iteration; convolutional and pooling layers for convolutional networks.

The Keras-RL module allows users to define reinforcement learning environments and agents or use existing ones to make tests. Environments set the way rewards are calculated and contain the observation spaces, the list of states, the list of actions and the policies used. Agents train and test a Keras model for a given environment.

3.7 Related Work: Reinforcement Learning with Deep Learning

Haaswijk, Collins and Seguin (2017) approaches the logic optimization problem with a reinforcement learning game. Boolean functions are represented by Majority Inverter Graphs (MIG) to which a finite set of transformations is defined. During the game, the MIGs are treated as the states s, and the valid transformations for the MIG from state s are treated as the actions a from state s. Choosing an action a in a state s_t is equivalent to transforming the MIG so that

$$s_t \xrightarrow{a} s_{t+1}. \tag{3.11}$$

The goal of the game is to obtain the optimal MIG, therefore a $score(s)$ function is defined to evaluate the MIG from state s according to a criterion, the MIG's logic depth or its number of nodes. The reward function r is given by $r(s_t, a_t) = score(s_t) - score(s_{t-1})$. Finally, the game can be played maximizing the total reward r_t, shown in equation 3.12.

$$\sum_{t=0}^{n} r_t = \sum_{t}^{n-1}(score(s_{t+1}) - score(s_t)) = score(s_n) - score(s_0) \tag{3.12}$$

A gradient policy is used in order to reward positively the actions that reduce the logic depth or number of nodes and negatively the actions that do otherwise. The model used to implement the policy is deep convolutional neural network. This network filters information from the MIG's adjacency matrix and uses the ReLU as activation function.

The method proposed by Haaswijk, Collins and Seguin (2017) is the first to apply machine learning in logic synthesis. The method had similar or even superior performance in several cases in comparison with state of the art algorithms for logic optimization.

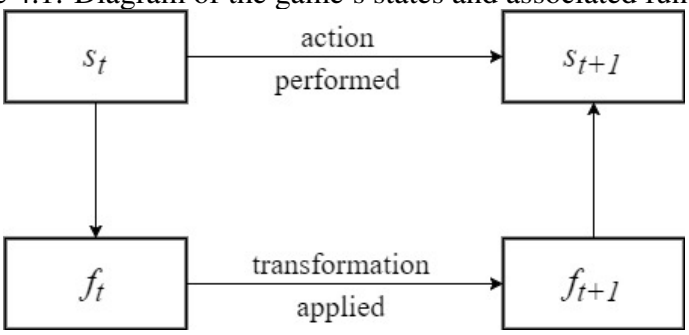
4 PROPOSED METHOD

This work proposes to solve NPN-matching with a game of reinforcement learning inspired on the canonical-based methods for determining boolean function equivalence. Boolean functions are represented by truth-table integers that correspond to the states of the game. The game's actions are the set of possible transformations that can be applied to a boolean function

$$T = \{flips,\ swaps,\ complement\ output\} \tag{4.1}$$

according to the NPN matching, as shown in table 2.3. As well as in (HAASWIJK; COLLINS; SEGUIN, 2017), choosing action a for a state s_t is equivalent to applying the transformation associated to action a to the boolean function associated to state s_t in order to obtain a new boolean function, which is associated to state s_{t+1}, as shown in figure 4.1.

Figure 4.1: Diagram of the game's states and associated functions.



An instance of the game is a finite sequence of states

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} s_n. \tag{4.2}$$

going from s_0, which corresponds to the input function, to s_n, which is associated to the function chosen to be the semi-canonical representative of the input function. Therefore, the goal of the game is finding such sequence, the fewer actions performed, the better.

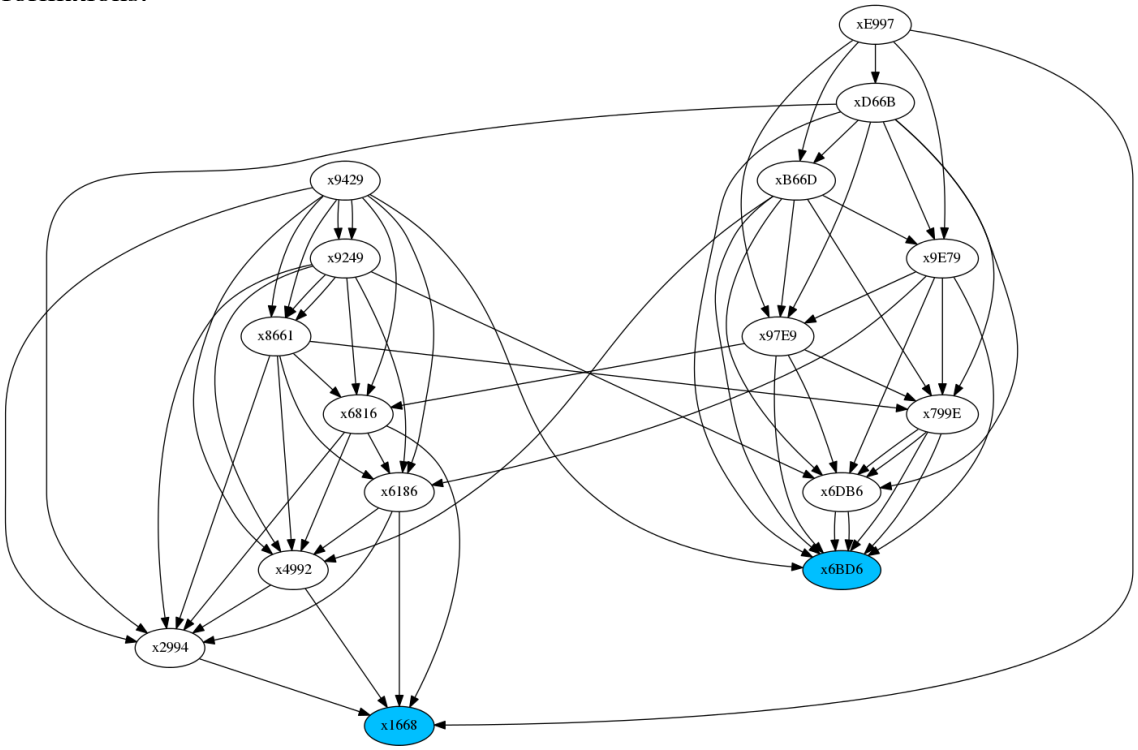
4.1 New Semi-canonical Form

A new semi-canonical form is proposed by this work. In this form, a representative function is a function f_t to which no single transformation, as shown in equation 4.1, generates a function f_{t+1} whose integer truth-table is lower than that of f_t. In other words,

only transformations that lower the function's integer truth-table can be applied. Taking this information into the context of the game means that a new rule must be added. The only valid actions for a given state are the ones that lower the integer truth-table of the state's associated function. This way, a final state is a state whose action set is empty.

Example: Given the set of 4-variable boolean functions that belong to the exact canonical form NPN-equivalence class whose lower representative is the 0x1668 function, figure 4.2 shows all valid transformations according to the semi-canonical form proposed by this work.

Figure 4.2: All valid transformations according to the proposed semi-canonical form for a 4-variable NPN equivalence class. The nodes are the states and the edges are the transformations.



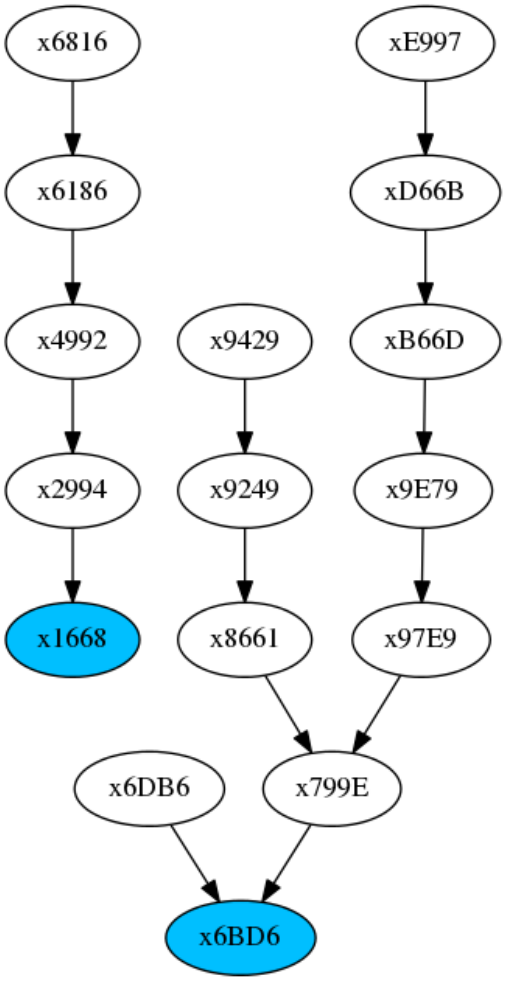
Notice that figure 4.2 shows two final states. One of them, the 0x6BD6, is not associated with the representative according to the exact NPN canonical form. This is a consequence of using a semi-canonical form, which is more relaxed than an exact canonical form. A set of functions that belong to the same equivalence class may be split in several smaller groups, each with its own representative. That is, a semi-canonical representative is not forcefully the lowest integer function of the set. Notice also that some of the functions have paths to both representatives. This is an issue for the canonicity of this form.

4.1.1 On the Canonicity of the New Semi-Canonical Form

Due to the fact that there may be several valid transformations for a given function, different sequences of transformations could be applied to the function, leading to distinct representatives. In order to guarantee that the sequence of transformations performed on a function is always the same, a deterministic algorithm to choose the transformation is needed.

Example: Figure 4.3 shows the equivalence class of the 0x1668 function allowing one single transformation per state.

Figure 4.3: Only a single transformation per function according to the proposed semi-canonical form for a 4-variable NPN equivalence class. The nodes are the states and the edges are the transformations.



Applying a deterministic algorithm to the equivalence class of the 0x1668 function limits the number of outgoing edges of each node to exactly one, except for the nodes of the final states. Notice also that this limitation splits the graph in two, one for each representative, eliminating ambiguity.

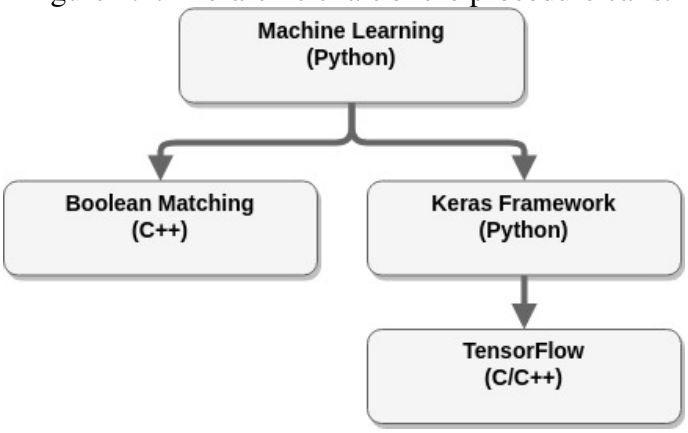
4.2 Implementation

The machine learning part of the game is implemented in Python and uses the Keras and Keras-RL frameworks, which call code written in C. The Python code also calls the boolean matching part through a wrapper. The wrapper creates a Python module from the C++ code and also adds routines for data type conversion. The data type conversions used are shown in table 4.1. Figure 4.4 shows a hierarchic chart of the procedure calls.

Table 4.1: C++ to Python and Python to C++ used type conversions.

C++ Type	Python Type
uint8_t (char)	integer
std::vector<T>	list
std::string	string

Figure 4.4: Hierarchic chart of the procedure calls.

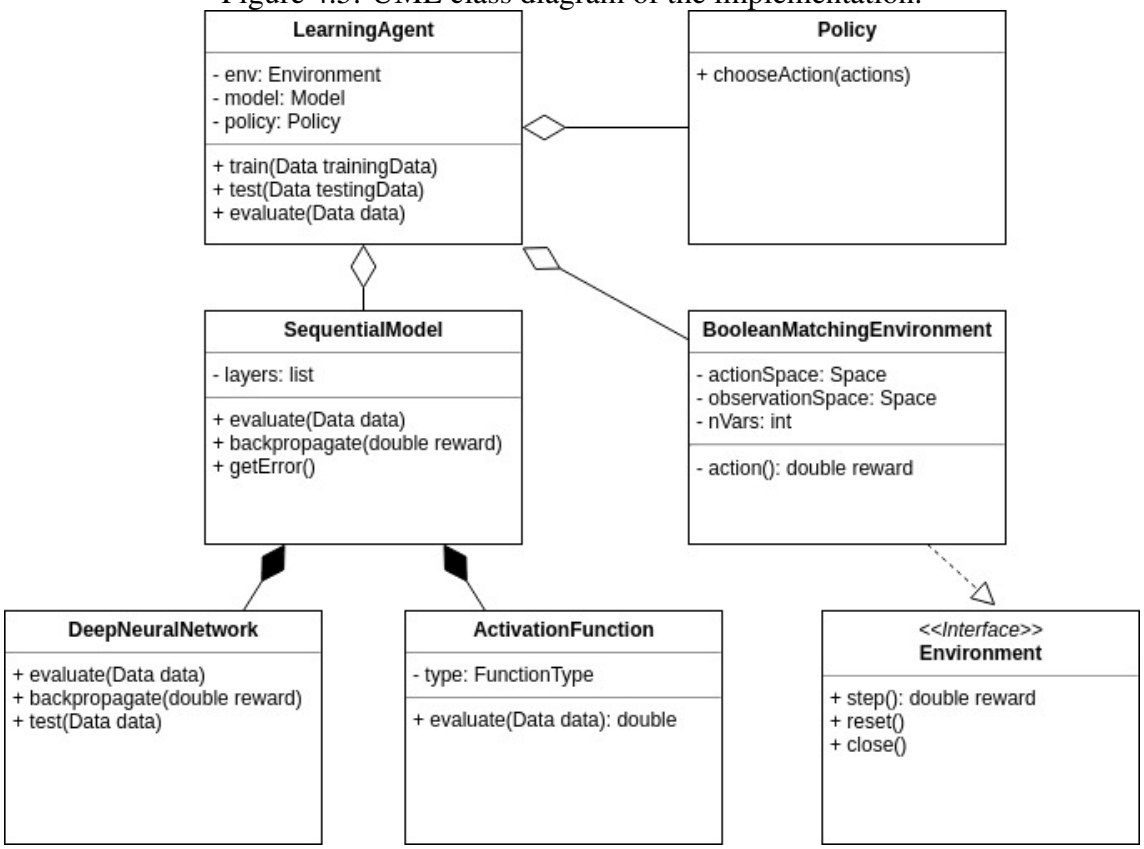


The boolean matching part of the game is implemented in C++. Three main operations are implemented: flips, swaps and complementing the output. There is also a secondary operation that checks whether or not the application of one of the main operations has lowered the integer truth-table of the input function and outputs a reward accordingly. This secondary operation is called reward.

4.2.1 Machine Learning Implementation

The machine learning step is implemented following the Object Oriented paradigm. Figure 4.5 shows an UML class diagram of the implementation. From a deep neural network and an activation function, a sequential model is built. The model is trained and tested by a learning agent for an environment defined for the boolean matching problem.

Figure 4.5: UML class diagram of the implementation.



- The *LearningAgent* controls the flow of the learning and the testing phases.
- The *DeepNeuralNetwork* is a Dense layer of 64 neurons.
- The *ActivationFunction* is a *ReLU*, as shown in equation 3.3.
- The *SequentialModel* simply concatenates the *DeepNeuralNetwork* and the *ActivationFunction*.
- The Policy selects the actions to be performed. Its method *chooseAction()* returns a list of actions ordered from highest probability of yielding a positive reward to lowest.
- The *BooleanMatchingEnvironment* makes the calls to the boolean matching part and defines both the action and the observation spaces. The action space is an enumeration of all actions. Equation 4.3 shows the action space for a 4-variable function, where $V = \{v1, v2, v3, v4\}$ is the function's support. Given $B = \{0, 1\}$ and the number of variables n, the observation space is B^{2^n}. Equation 4.4 shows an

observation for a 4-variable function.

$$
\begin{aligned}
&A = \{ \\
&flip(v1), \\
&flip(v2), \\
&flip(v3), \\
&flip(v4), \\
&swap(v1, v2), \\
&swap(v1, v3), \\
&swap(v1, v4), \\
&swap(v2, v3), \\
&swap(v2, v3), \\
&swap(v3, v4), \\
&complement\_output \\
&\}
\end{aligned}
\tag{4.3}
$$

$$observation\_example = (0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0) \tag{4.4}$$

4.2.2 Boolean Matching Implementation

The main part of the boolean matching implementation is a function, called *action*, that performs an action and returns the reward. The pseudo-code for *action* is shown in algorithm 1. The *action* procedure takes as input a list of actions ordered from highest probability of yielding a positive reward to lowest and the maximum number of actions it should attempt to perform looking for a positive reward.

The data structure used to represent boolean functions is an array of bits that correspond to the function's truth-table. The bits of the array are implemented as the *uint8_t* type so that they can be easily converted to integers in Python as the array turns into a list using the wrapper's supported type conversions.

Algorithm 1 Action algorithm

```
1:  procedure ACTION(actions, maxAttempts)
2:      counter ← 0
3:      reward ← 0
4:      for all action in actions do
5:          singleAction(action)
6:          reward ← reward()          ▷ rewards positively if the integer was lowered
7:          counter ← counter + 1
8:          if reward > 0 or counter > maxAttempts then
9:              return reward
10:         end if
11:         undoAction(action)
12:     end for
13: end procedure
14:
15: procedure SINGLEACTION(action)
16:     if isFlip(action) then
17:         v1 ← getFlipVariable(action)
18:         flip(v1)
19:     else
20:         if isSwap(action) then
21:             v1, v2 ← getSwapVariables(action)
22:             swap(v1, v2)
23:         else
24:             complementOutput()
25:         end if
26:     end if
27: end procedure
```

5 RESULTS AND DISCUSSION

The method was implemented in C++ 11 and Python 3 and results were run on a i5-2400 CPU @ 3.10 GHz machine with 8 GB of RAM. After the execution, the output functions were uniquified and the number of distinct functions was counted. This way, the number of equivalence classes was determined.

5.1 Performance x Quality

The metric for Performance x Quality used in this work is shown in (HUANG et al., 2013). In this metric, performance is evaluated as the runtime in seconds, the faster the better. Quality is the number of equivalence classes found, the fewer the better. Since this work evaluates a semi-canonical form, the quality metric is important to estimate how the final result may be affected. However, the main concern with a semi-canonical form is establishing a trade-off between performance and quality in which performance is highly improved whilst quality is slightly diminished. Therefore, the goal when using a semi-canonical form is firstly increasing performance and secondly maintaining quality.

Tables 5.1, 5.2, 5.3 show a performance x quality comparison for full, partial and non DSD functions of 6 to 14 variables. The sets of functions were extracted from the benchmarks available at (MISHCHENKO, 2014), which are benchmarks of practical functions. These functions have been chosen because they are commonly used in industry according to the authors of (MISHCHENKO, 2014). The results for (HUANG et al., 2013) were obtained from the implementation available in the ABC Tool by running the command *testnpn -A 4 <inputfile>*.

Table 5.1: Performance x Quality for full DSD functions.

		Proposed Method		**(HUANG et al., 2013)**	
N	**Functions**	**Eq. Classes**	**T (s)**	**Eq. Classes**	**T (s)**
6	10k	116	19.89	59	0.01
8	10k	742	63.45	383	0.02
10	10k	1308	313.55	855	0.05
12	10k	1655	432.37	1179	0.13
14	10k	2438	582.65	890	0.44

Results show that the proposed method in this implementation had worse results in performance and in quality for all benchmarks tested. Notice that the partial DSD functions take slightly longer time than the DSD, and non-DSD are even more time con-

Table 5.2: Performance x Quality for partially DSD functions.

		Proposed Method		**(HUANG et al., 2013)**	
N	**Functions**	**Eq. Classes**	**T (s)**	**Eq. Classes**	**T (s)**
6	10k	787	29.02	411	0.01
8	10k	2471	115.19	1679	0.03
10	10k	3352	473.8	2289	0.07
12	10k	4518	535.24	2966	0.19
14	10k	4846	582.29	2474	0.62

Table 5.3: Performance x quality for non-DSD functions.

		Proposed Method		**(HUANG et al., 2013)**	
N	**Functions**	**Eq. Classes**	**T (s)**	**Eq. Classes**	**T (s)**
6	10k	433	45.99	189	0.01
8	10k	2423	127.63	1396	0.04
10	10k	1730	526.52	772	0.09
12	10k	1904	550.37	860	0.2
14	10k	1622	591.86	945	0.79

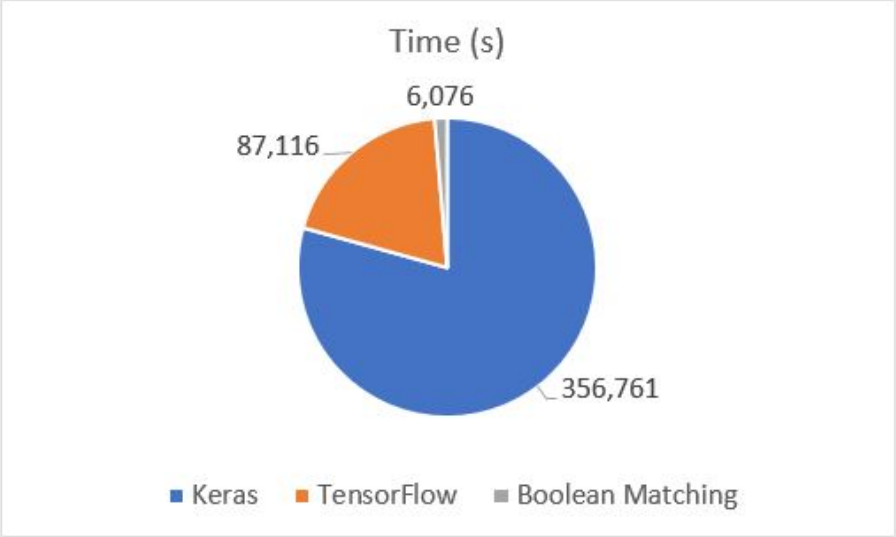
suming to run for both methods. Also, regarding the proposed method, even though the runtime for up to 12 variables may vary throughout the different benchmarks, the runtime for 14 variables is very similar. This can be explained by the fact that, as the number of variables increases, so does the size of the truth-table. Consequently, the size of the input of the neural network rises. The bigger the size of the input, more neurons have to be used in order to predict the action to be performed, up until a point where almost the entire network is activated simultaneously. This could be further analyzed if benchmarks for 16-variable functions were run. However, there was not enough memory to do so in the 8 GB RAM system used.

5.2 Time Consuming Parts

With the intent of measuring which parts of the implementation take the longest to run, a profile of the runtime was made, as shown in figure 5.1, using the $cProfile$ module from Python. Notice that the TensorFlow and the Keras modules, which implement the machine learning part of this method, represent the great majority of the runtime. If the machine learning part were entirely implemented in a compiled language and if it were run on Tensor Processing Units (TPU), the runtime could be greatly reduced. Furthermore, if the machine learning implementation were adapted so that the boolean matching part would use a more efficient data structure, the boolean matching implementation performance would also be improved. The current implementation uses

a $std :: vector < uint8\_t >$, in which each $uint8\_t$ stores a single bit. A much more efficient data structure found in the ABC Tool is an array of $uint64\_t$ of length 1024 that can represent functions of up to 16 variables. This way, boolean function transformations can be applied with bitwise operators to reduce the number of instructions executed to perform a transformation.

Figure 5.1: Profile of the runtime for training full DSD 6-variable functions.



5.3 Train Exhaustively Once, Perform Better Always

An advantage of using machine learning is being able to spend a long time training the model once and later run it as many times as necessary. An iteration of the learning process is called learning step, i.e., a learning step corresponds to applying an action, calculating the reward and updating the weights. Table 5.4 shows that, when the number of attempts is fixed, increasing the time spent on training improves the quality of results without significantly changing the runtime. Table 5.5 shows that, when the number of attempts is unlimited, increasing the number of learning steps reduces the average number of attempts to perform an action until a positive reward is found. In other words, the training time can be increased to either improve the quality of the results without affecting runtime or improve runtime without changing the quality of results.

Table 5.4: Time spent on training x quality for all 4 variable functions with $maxAttempts = 5$.

Learning Steps	Training Time (s)	Runtime (s)	Eq Classes
1000	8.92	198.59	4026
2000	16.68	214.48	3627
4000	31.91	213.97	2744
8000	65.21	211.09	2607
16000	132.15	241.7	1473
32000	259.5	205.79	1329
64000	518.22	243.06	1136
128000	1031.05	233.69	1099
256000	2008.53	215.54	1066

Table 5.5: Learning steps x number of attempts until finding a positive reward transformation for 10k full DSD 6-variable functions with $maxAttempts = \infty$.

Learning Steps	Attempts on Average	Runtime (s)
10000	5.509696	54.20
25000	4.100001	46.28
50000	3.525992	39.11
100000	3.030859	19.86

5.4 The Cost of Using Machine Learning

Applying machine learning to solve a given problem has a cost both in performance and in the form of addition of constraints to the implementation. These costs have to be considered when choosing machine learning as the solution to the problem. The cost in performance comes from the fact that, for every decision taken by the method, the machine learning model has to be reevaluated. Depending on the size of the model and on how many decisions the method has to make for every instance of the problem, this can mean a lot of operations. The cost in addition of constraints to the implementation comes from the form of the inputs and outputs of a neuron. This form adds restrictions to the interface between the problem and the machine learning technique.

5.4.1 The Interface Between Machine Learning and Boolean Matching

As discussed in chapter 3, a neuron's inputs and output are a real number in the interval $[-1, 1]$ or $[0, 1]$. There are two main strategies of encoding information into the inputs of a neural network. Either a lot of information is encoded into an input, which would translate to describing several bits of a truth-table with a single variable, or just a little of information per input, which would mean a single bit per variable. The more

information is encoded in a single input, the harder it is for the machine learning model to distinguish between different values, thus decreasing its accuracy. Therefore, there are certain constraints on how information can be inserted into a neural network.

Considering these constraints, there are also two strategies to adjust the interface between boolean matching and machine learning. Either a data conversion is applied every time the neural network runs or the data structure is conceived to work directly with the network without the need for conversion or cast. This work has chosen to use a data structure that works directly with the neural network. However, this has had an impact in performance because, as discussed in section 5.2, much more efficient data structures could have been used.

5.5 The Issue with the Rewarding Mechanism

For many applications including boolean matching, there is no unexpensive way of checking whether a solution is a global optimum. This poses a problem for reinforcement learning methods because it is hard to train a model to attain a certain state whose given reward in the training phase is no different than the reward of many other states. There is no way of detecting if the lowest representative has been attained without running an NPN canonical classifier. This is why the proposed method has had to use a semi-canonical form instead of a canonical form. Also, searching for the global optimum would only be possible if the reinforcement learning part of the proposed method could accept negative reward actions. However, if negative rewards were to be accepted without checking for global optimums, there would be no halting condition for the method.

6 CONCLUSION

This work proposed a new method for boolean matching using machine learning. This is to our knowledge the pioneer of applying machine learning to boolean matching, following the work of (HAASWIJK; COLLINS; SEGUIN, 2017), which is the pioneer of applying machine learning to logic synthesis. Both methods use a reinforcement learning approach combined with deep neural networks. This work also proposes a new semi-canonical form.

Results showed that the proposed method has had worse quality of results and worse performance than the compared method. However, performance can be improved by changing the implementation language and data structures as well as using specific TPUs in hardware. Results also showed that increasing the training time can either improve the quality of results or the performance.

This work discussed the intrinsic cost of using machine learning and the consequences of adapting a problem to be solved by reinforcement learning techniques. Furthermore, an issue with the rewarding mechanism for problems in which checking if the solution is the global optimum is expensive has been discussed.

REFERENCES

ABDOLLAHI, A. Signature based boolean matching in the presence of don't cares. In: **2008 45th ACM/IEEE Design Automation Conference**. [S.l.: s.n.], 2008. p. 642–647. ISSN 0738-100X.

ABDOLLAHI, A.; PEDRAM, M. Symmetry detection and boolean matching utilizing a signature-based canonical form of boolean functions. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 27, n. 6, p. 1128–1137, June 2008. ISSN 0278-0070.

ASSODIKY, H.; SYARIF, I.; BADRIYAH, T. Deep learning algorithm for arrhythmia detection. In: **2017 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC)**. [S.l.: s.n.], 2017. p. 26–32.

BAILEY, B. Eda challenges machine learning. **Semiconductor Engineering**, dec. 2017. Available from Internet: <https://semiengineering.com/eda-challenges-machine-learning/>.

BOOLE, G. **Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities**. [s.n.], 1853. Also available from Dover, New York 1958, ISBN 0-486-60028-9. Available from Internet: <http://www.gutenberg.org/etext/15114>.

HAASWIJK, W.; COLLINS, E.; SEGUIN, B. Deep learning for logic optimization. In: **2017 International Workshop on Logic Synthesis (IWLS)**. [S.l.: s.n.], 2017.

HUANG, Z. et al. Fast boolean matching based on npn classification. In: **2013 International Conference on Field-Programmable Technology (FPT)**. [S.l.: s.n.], 2013. p. 310–313.

KAGGLE. **Feature Engineering**. 2017. Https://s3.amazonaws.com/dq-content/186/overfitting.svg. Acessado em 23/12/2017.

KAPOOR, B. Improved technology mapping using a new approach to boolean matching. In: **Proceedings the European Design and Test Conference. ED TC 1995**. [S.l.: s.n.], 1995. p. 86–90.

KATEBI, H.; MARKOV, I. L. Large-scale boolean matching. In: **2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)**. [S.l.: s.n.], 2010. p. 771–776. ISSN 1530-1591.

LAU, M. M.; LIM, K. H. Investigation of activation functions in deep belief network. In: **2017 2nd International Conference on Control and Robotics Engineering (ICCRE)**. [S.l.: s.n.], 2017. p. 201–206.

MAILHOT, F.; MICHELI, G. D. Technology mapping using boolean matching and don't care sets. In: **Proceedings of the European Design Automation Conference, 1990., EDAC.** [S.l.: s.n.], 1990. p. 212–216.

MISHCHENKO, A. **Full, partial and non DSD practical function benchmarks.** 2014. Accessed: 2018-05-16. Available from Internet: <http://people.eecs.berkeley.edu/~alanmi/temp5/>.

NGUYEN, N. D.; NGUYEN, T.; NAHAVANDI, S. System design perspective for human-level agents using deep reinforcement learning: A survey. **IEEE Access**, v. 5, p. 27091–27102, 2017.

PETKOVSKA, A. et al. Fast hierarchical npn classification. In: **2016 26th International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.: s.n.], 2016. p. 1–4.

SILVER, D. et al. Mastering the game of go with deep neural networks and tree search. **Nature**, Nature Publishing Group, a division of Macmillan Publishers Limited. All Rights Reserved. SN -, v. 529, p. 484 EP –, Jan 2016. Article. Available from Internet: <http://dx.doi.org/10.1038/nature16961>.

SWINGLER, K. **Applying Neural Networks: A Practical Guide**. Academic Press, 1996. ISBN 9780126791709. Available from Internet: <https://books.google.com.br/books?id=bq0YnP4BNKsC>.

WANG, S. **Artificial neuron and method of using same**. Google Patents, 1995. US Patent 5,390,136. Available from Internet: <https://www.google.com/patents/US5390136>.

YANG, W.; WANG, L.; MISHCHENKO, A. Lazy man's logic synthesis. In: **2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2012. p. 597–604. ISSN 1092-3152.

ZHANG, J. et al. A canonical-based npn boolean matching algorithm utilizing boolean difference and cofactor signature. **IEEE Access**, v. 5, p. 27777–27785, 2017.

ZHANG, J. et al. An efficient NPN boolean matching algorithm based on structural signature and shannon expansion. **CoRR**, abs/1708.04597, 2017. Available from Internet: <http://arxiv.org/abs/1708.04597>.

Boolean Matching com Aprendizagem de Máquina

Réges Eduardo Oberderfer Júnior[1], **André Inácio Reis**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{reojunior,andreis}@inf.ufrgs.br

Abstract. *Boolean matching is the task of determining equivalence between boolean functions, a key step in technological mapping. This work proposes a new method to solve boolean matching using machine learning. A reinforcement learning approach is used to match boolean functions recasting matching as a game. The game's objective is to find a canonical representative function, which is the function that corresponds to the smallest integer in the equivalence class. If the output of the method is the same for two functions, they are equivalent.*

Resumo. *Boolean matching é a tarefa de determinar a equivalência entre funções booleanas, uma etapa essencial no mapeamento tecnológico. Este trabalho explora um novo método para resolver boolean matching usando aprendizagem de máquina. Boolean matching é aplicado usando um jogo de aprendizagem por reforço. O objetivo do jogo é encontrar a função representante canônica, que é tomada como a função que corresponde ao menor inteiro na classe de equivalência. Se para duas funções a mesma função resultante é obtida, elas são equivalentes.*

1. Introdução

Determinar se duas funções booleanas são equivalentes independentemente de permutação e negação de entradas e de negação da saída (NPN) é um problema muito importante na área da síntese lógica. A síntese lógica é uma das etapas no fluxo standard cell em VLSI. O mapeamento tecnológico, na fase dependente de tecnologia da síntese lógica, consiste em buscar portas lógicas definidas numa biblioteca para implementar funções booleanas num circuito [Mailhot and Micheli 1990]. Essa busca deve ser capaz de encontrar a equivalência entre funções NPN. Duas funções f_1 e f_2 são NPN-equivalentes se é possível transformar f_1 em f_2 usando permutação e/ou negação de entradas e/ou negação da saída. Boolean matching é a tarefa de determinar a equivalência entre funções booleanas, uma tarefa essencial no mapeamento tecnológico [Kapoor 1995].

Existem diversos métodos para realizar Boolean matching. Alguns deles se baseiam em formas canônicas de representação de modo que, se ambas as funções f_1 e f_2 resultam num mesmo representante canônico, elas são equivalentes [Abdollahi and Pedram 2008][Petkovska et al. 2016][Zhang et al. 2017a]. Outros métodos buscam a relação de correspondência entre as variáveis das duas funções. Dessa forma, esses métodos encontram a transformação que deve ser aplicada a uma das funções para que a outra seja obtida [Abdollahi 2008][Zhang et al. 2017b]. Existem ainda algoritmos baseados em SAT, que transformam o problema de Boolean matching em uma fórmula booleana [Katebi and Markov 2010].

1.1. Motivação

Nos últimos anos, aprendizagem de máquina, que era uma simples ferramenta para reconhecimento de padrões, tornou-se capaz de vencer um humano em um jogo de Go [Silver et al. 2016]. Várias tarefas como controle de membros robóticos [Mnih et al. 2015] são realizadas através da aprendizagem por reforço que foi revolucionada por avanços recentes em $deep\ learning$. O sucesso desse método está na capacidade de encontrar automaticamente características especiais e construir modelos hierárquicos para um dado problema. Isso só é possível devido ao uso combinado de aprendizagem por reforço e redes neurais.

Empresas que trabalham com EDA vêm investindo em aprendizagem de máquina [Bailey 2017]. O seu interesse é tomar proveito desse método automático para acelerar o processo de convergência do fluxo de projeto de circuitos integrados. Devido à importância do Boolean matching na fase de mapeamento tecnológico e ao interesse das empresas de EDA, este trabalho explora um novo método para resolver Boolean matching usando aprendizagem de máquina.

1.2. Objetivo

O objetivo deste trabalho é expandir o estudo da aplicação de métodos de aprendizagem de máquina em síntese lógica, iniciado por [Haaswijk et al. 2017], que usa deep learning para otimização lógica. Note que o trabalho proposto em [Haaswijk et al. 2017] é pioneiro no uso de aprendizagem de máquina para síntese lógica, e esta área ainda está começando a ser explorada. Assim acreditamos que nosso trabalho será pioneiro na área matching de células de biblioteca usando técnicas de aprendizagem de máquina, e a experiência adquirida durante o trabalho será importante para o entendimento do potencial de aplicação de aprendizagem de máquina em outros problemas de síntese lógica.

1.3. Organização deste Trabalho

Este trabalho é organizado da seguinte maneira: a seção 2 apresenta os conceitos básicos; a seção 3 discute trabalhos relacionados; a seção 4 apresenta o método proposto por este trabalho; a seção 5 mostra o cronograma de atividades.

2. Conceitos

2.1. Funções Booleanas

Dado $B = \{0, 1\}$ e $O = B \cup \{\mathbf{X}\}$, uma função booleana f de n entradas é uma função $f : B^n \rightarrow O$, onde **X** é o símbolo que representa $don't\ care$. Uma função booleana $completamente\ especificada$ permite apenas valores $\{0, 1\}$ como saída.

Uma das maneiras mais usadas para representar uma função booleana é a $tabela\ verdade$, que lista os valores de saída de uma função f para todas as possíveis combinações de entrada. Um exemplo de tabela verdade é mostrado na tabela 1.

Assumindo que a ordem das variáveis de entrada de f é $[a, b]$, f pode ser representada também apenas pela última coluna de sua tabela verdade interpretada como um número. **Exemplo:** $f = a + b = 0111_2 = 7_{16}$.

Tabela 1. Exemplo de tabela verdade para a função $f = a + b$.

a	b	f
0	0	0
0	1	1
1	0	1
1	1	1

2.1.1. Cofator de uma Função Booleana

Dada uma função booleana f com entradas $[e1, e2, ..., e_n]$, o $cofator$ de f para uma variável de entrada e_i é a avaliação de e_i na função f [Boole 1853], tal que

$$f_{e_i} = f(e_i = 1) \quad (1)$$

$$f_{\overline{e_i}} = f(e_i = 0) \quad (2)$$

O cofator $positivo$ de f para a variável e_i é f_{e_i}, como mostrado na equação 1. O cofator $negativo$ de f para a variável e_i é $f_{\overline{e_i}}$, como mostrado na equação 2.

2.1.2. Variáveis Unate e Binate

As propriedades $unate$ e $binate$ são características importantes para analisar o comportamento de variáveis numa função booleana. Dados os cofatores positivo e negativo de f para uma variável de entrada e_i, o comportamento de e_i na função f é mostrado na tabela 2.

Tabela 2. Comportamento unate e binate de variáveis booleanas.

Relação	Comportamento
$f_{e_i} \equiv f_{\overline{e_i}}$	$don't\ care$
$(f_{e_i} \neq f_{\overline{e_i}}) \wedge (f_{e_i} \equiv f_{e_i} + f_{\overline{e_i}})$	$positive\ unate$
$(f_{e_i} \neq f_{\overline{e_i}}) \wedge (f_{\overline{e_i}} \equiv f_{e_i} + f_{\overline{e_i}})$	$negative\ unate$
$(f_{e_i} \neq f_{\overline{e_i}}) \wedge (f_{e_i} \neq f_{e_i} + f_{\overline{e_i}}) \wedge (f_{\overline{e_i}} \neq f_{e_i} + f_{\overline{e_i}})$	$binate$

2.1.3. Classes de Equivalência de Funções Booleanas

Duas funções booleanas pertencem a uma mesma classe de equivalência quando é possível aplicar uma série de transformações de modo que partindo da primeira função a segunda seja obtida. As transformações possíveis são permutação de variáveis de entrada, complemento de variáveis de entrada e complemento da saída. A tabela 3 mostra a nomenclatura das classes de equivalência.

Se duas funções booleanas f_A e f_B pertencem à mesma classe P, NP, PN ou NPN, elas são P, NP, PN ou NPN-equivalentes, respectivamente.

Exemplo: Considere as funções $f_1 = a * b + c$, $f_2 = a + b * c$ e $f_3 = \overline{a} + b * c$, $f_4 = \overline{a + b * c}$ e $f_5 = \overline{\overline{a} + b * c}$. A tabela 4 mostra a equivalência entre elas.

Tabela 3. Classes de Equivalência de Funções Booleanas

Transformações permitidas	Classe de equivalência
Permutação das entradas	P
Complemento e permutação das entradas	NP
Permutação das entradas e complemento da saída	PN
Complemento e permutação das entradas e complemento da saída	NPN

Tabela 4. Exemplo de Equivalência entre Funções Booleanas

Funções	Equivalência	Transformações
f_1 e f_2	P, NP, PN e NPN	Permutar a com c
f_1 e f_3	NP e NPN	Permutar a com c e complementar a
f_1 e f_4	PN e NPN	Permutar a com c e complementar a saída
f_1 e f_4	NPN	Permutar a com c, complementar a e complementar a saída

2.1.4. Boolean Matching

Boolean matching é um problema que consiste em determinar se duas funções booleanas são equivalentes. Para que duas funções booleanas sejam equivalentes, elas devem pertencer à mesma classe. Assim, P-matching, NP-matching, PN-matching e NPN-matching são instâncias do problema de boolean matching para cada uma das respectivas classes de equivalência de funções.

2.2. Neurônios Artificiais

Os neurônios artificiais modelam matematicamente o funcionamento de neurônios biológicos. Seu comportamento consiste em gerar uma saída numérica a partir de entradas numéricas ponderadas por pesos.

A notação usada para neurônios artificiais e redes neurais neste trabalho é mostrada na tabela 5. Um modelo de um neurônio artificial, onde $[x_1, x_2, x_3, ..., x_n]$ são as suas entradas, pode ser visto na figura 1.

Tabela 5. Notação para neurônios.

Símbolo	Descrição
n	número de neurônios na camada atual
i	neurônio i de uma camada
j	neurônio j da camada seguinte à camada de i
w_{ij}	peso da aresta que liga o neurônio i ao neurônio j
o_j	saída do neurônio j
v_j	entrada do neurônio j
d_j	saída esperada do neurônio j
e_j	derivada do erro do neurônio j
f	função de ativação
f'	derivada da função de ativação
α	taxa de aprendizagem

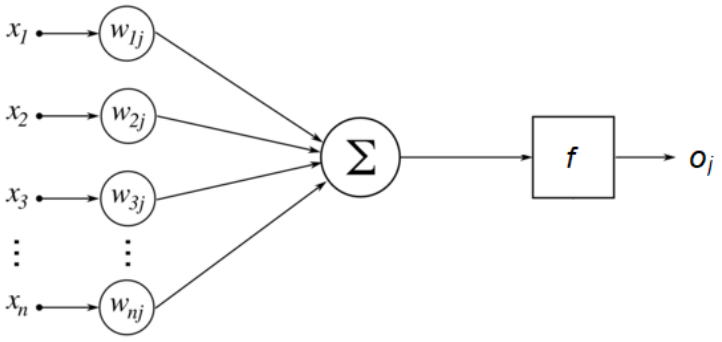


Figura 1. Modelo de neurônio artificial. [Wang 1995]

2.2.1. Funções de Ativação

Uma função de ativação é uma função que toma como entrada a soma das entradas do neurônio multiplicadas pelos pesos e devolve um valor entre $[0, 1]$ ou $[-1, 1]$. Além disso, as funções de ativação também providenciam um comportamento não linear que permite a aproximação de qualquer função [Lau and Lim 2017]. A forma genérica das funções de ativação é mostrada na equação 3. As equações 4, 5, 6 e 7 mostram, respectivamente, as funções de ativação **Heaviside**, **ReLU**, **Logística** e **Tanh**.

$$o_j = f(\sum_{i=1}^{n} w_{ij}\, o_i) \tag{3}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{4}$$

$$f(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases} \tag{5}$$

$$f(x) = \frac{1}{1 + \exp^{-x}} \tag{6}$$

$$f(x) = \frac{2}{1 + \exp^{-2x}} - 1 \tag{7}$$

2.2.2. Derivada do Erro da Saída de um Neurônio

O erro da saída de um neurônio pode ser simplesmente calculado como a diferença entre a sua saída e a saída esperada. A derivada do erro é dada pela equação 8.

$$e_i = f'(o_i)(d_i - o_i) \tag{8}$$

2.3. Redes Neurais

Uma rede neural é um modelo estatístico de um sistema do mundo real formado por várias camadas de neurônios. A rede usa parâmetros conhecidos como $pesos$ para propagar informação de um neurônio de uma camada para os neurônios da camada seguinte. Os dados são inseridos na camada de $entrada$ e propagados pelas camadas $ocultas$ até atingirem a camada de $saida$. Dessa forma, o modelo de rede neural mapeia valores de um conjunto de entrada para valores de um conjunto associado de saída. Os pesos fazem o ajuste da correta associação entrada-saída para cada um dos valores do conjunto de entrada. A escolha dos pesos é feita no $treinamento$ da rede neural por um processo chamado $backpropagation$, que propaga o erro das saídas para as camadas ocultas e usa a contribuição de erro de cada neurônio para melhorar o ajuste dos pesos a cada iteração do algoritmo.

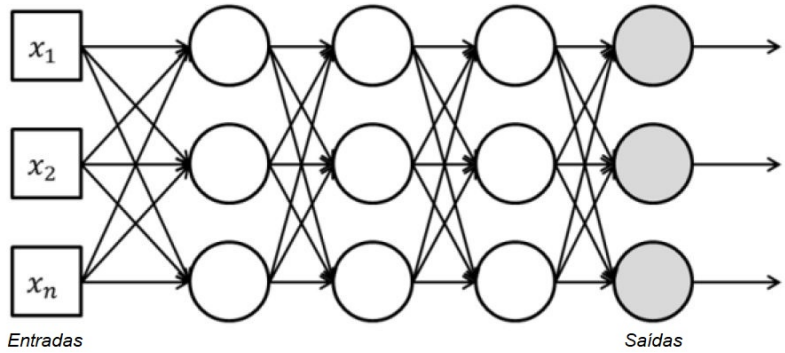


Figura 2. Modelo simples de uma Rede Neural. [Assodiky et al. 2017]

2.3.1. Treinamento de uma Rede Neural

As redes neurais fazem parte dos métodos de aprendizagem de máquina $supervisionados$. Para treinar uma rede neural, é preciso um conjunto de dados de treinamento. O conjunto de dados de treinamento pode ser visto como uma tabela em que cada coluna representa um $atributo$, e cada linha é uma instância, como mostrado na tabela 6. Alguns desses atributos, os $atributos\ alvo$, são aqueles para os quais se quer prever um valor para futuros dados. O treinamento da rede neural consiste em iterar sobre as instâncias da tabela, usando os atributos não-alvo como fonte de informação para as entradas da rede e os atributos alvo como as saídas. A cada iteração, os pesos são ajustados para melhor se adaptarem aos dados do conjunto de treinamento. Ou seja, o objetivo do treinamento é minimizar o $erro$ entre o valor dos atributos alvo das instâncias e o valor previsto pela rede.

Tabela 6. Um conjunto de dados de treinamento.

Instância	Atributo 1	Atributo 2	Atributo 3	Atributo Alvo 1	Atributo Alvo 2
x1	3	0.1	4	1	0.5
x2	7	0.3	6	0	1
x3	4	0.2	7	1	0

Além do conjunto de dados de treinamento, também é necessário um conjunto de dados de teste. O conjunto de dados de teste é uma parcela dos dados - em geral, 10% a 20% - que é separada dos dados iniciais e não é inserida no treinamento. Os dados de teste são utilizados para verificar se a rede não se adaptou bem demais aos dados de treinamento. A figura 3 mostra um exemplo em que o modelo foi sobreajustado aos dados de treinamento; esse fenônomeno é conhecido como $overfitting$. Um modelo sobreajustado aos dados pode ter erro muito grande ao prever valores para novos dados de entrada, pois o padrão modelado é mais complexo do que aquele seguido pelos dados. Uma solução para esse problema é verificar o erro da rede neural para o conjunto de dados de teste.

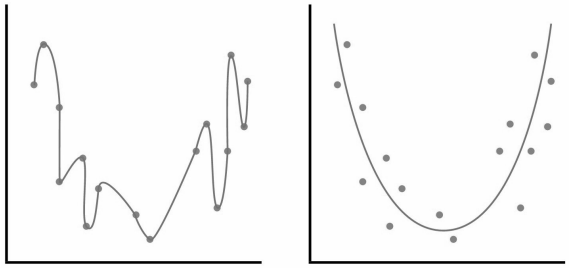


Figura 3. À esquerda, modelo sobreajustado aos dados (overfitting). À direita, modelo bem ajustado. [Kaggle 2017]

2.3.2. Algoritmo de Backpropagation

Numa rede neural, a derivada pode ser propagado da saída para as entradas usando o backpropagation. Inicialmente, calcula-se a derivada do erro para os neurônios da camada de saída seguindo a equação 8. Em seguida, as derivadas dos erros são propagadas usando a equação 9.

$$e_i = f'(o_i) \sum_{j=1}^{n} e_j \, w_{ij} \tag{9}$$

O método de backpropagation é dado pelo seguinte algoritmo [Swingler 1996].

1. Inicialize todos os pesos da rede neural com valores aleatórios entre $[0, 1]$ ou $[-1, 1]$
2. **Repita**
   (a) Escolha uma instância dos dados de treinamento.
   (b) Copie os valores dos atributos não-alvo para as entradas da rede neural.
   (c) Propague a entrada por todas as camadas da rede neural usando a função de ativação e os pesos.
   (d) Calcule a derivada do erro para a camada de saída.
   (e) Propague a derivada do erro das entradas para as saídas.
   (f) Atualize os pesos.
3. **Até que** o erro seja suficientemente pequeno.

2.3.3. Atualização dos Pesos

Para mudar os pesos de uma rede neural, aplicam-se a equações 10 e 11.

$$\Delta w_{ij} = \alpha \, e_j \, o_i \tag{10}$$

$$novo\_w_{ij} = w_{ij} + \Delta w_{ij} \tag{11}$$

2.4. Aprendizagem por Reforço

Aprendizagem por Reforço (AR) maximiza a longo prazo a recompensa obtida. Para fazer isso, AR busca uma política ótima π^* que mapeia cada estado s às probabilidades de cada ação a ser escolhida de modo a maximizar a recompensa r a longo prazo [Nguyen et al. 2017]. A função de mapeamento Γ_π é dada pela equação 12, onde $\Lambda_\pi(s)$ é o espaço de ações possíveis no estado s seguindo a política π.

$$\Gamma_\pi = \{P(s \xrightarrow{a} s'|\pi) : \forall a \in \Lambda_\pi(s)\} \tag{12}$$

3. Trabalhos Relacionados

3.1. Boolean Matching

3.1.1. Baseado em Representação Canônica

[Petkovska et al. 2016] usam um método hierárquico para determinar a equivalência NPN de funções booleanas. O método hierárquico guarda informações intermediárias do processo de matching que são reutilizadas posteriormente, aumentando o desempenho do método. As classes de equivalência são organizadas de modo hierárquico, e as transformações finais são obtidas transformando as funções representativas das classes intermediárias nas funções representativas das classes finais.

[Zhang et al. 2017a] apresentam um algoritmo para resolver o matching de funções booleanas completamente especificadas de uma única saída. Os autores descrevem vetores de assinatura por diferença booleana e por cofator. Os vetores de assinatura e propriedades de simetria das variáveis são usados para realizar o boolean matching com formas canônicas. As propriedades de simetria reduzem o espaço de busca e obter performance superior aos métodos propostos anteriormente.

3.1.2. Baseado em Assinaturas

[Abdollahi 2008] aborda o problema de boolean matching, usando assinaturas para determinar a equivalência de funções booleanas incompletamente especificadas pela primeira vez. O método usa BDDs para representar as funções, um para o seu on-set e um para o seu off-set. Assinaturas extraídas dos BDDs são usadas para criar um grafo bipartite. O grafo é reduzido e um algoritmo de branch and bound é aplicado para encontrar as transformações do matching.

[Zhang et al. 2017b] apresentam um método que usa um vetor de assinaturas estrutural para efetuar boolean matching. Propriedades de simetria são usadas para reduzir

o espaço de busca. O algoritmo é baseado numa estratégai de decomposição recursiva e busca.

3.1.3. Baseado em SAT

[Katebi and Markov 2010] mostram um método de boolean matching que usa AIGs, simulação e SAT. AIGs são usados no lugar de BDDs porque o método funciona para circuitos de larga escala. Uma instância de SAT é construída a partir da XOR $f \oplus g$. Se a fórmula booleana CNF for satisfatível, as f e g não são equivalentes.

3.2. Aprendizagem por Reforço Combinado com Deep Learning

[Haaswijk et al. 2017] aborda o problema de otimização lógica como um jogo de aprendizagem por reforço. As funções booleanas são representadas por Majority Inverter Graphs (MIG) para os quais é definido um conjunto finito de transformações. No contexto do jogo, os MIGs são tratados como estados s, e as transformações válidas para o MIG do estado s são tratadas como as ações a possíveis no estado s. Escolher a ação a em um estado s_t corresponde a transformar o MIG de modo que

$$s_t \xrightarrow{a} s_{t+1}. \tag{13}$$

O objetivo do jogo é obter o MIG ótimo, por isso os autores definem uma função $score(s)$ que avalia o estado s segundo um critério de profundidade lógica ou tamanho do MIG. A função recompensa r é definida por $r(s_t, a_t) = score(s_t) - score(s_{t-1})$. Dessa forma, o jogo pode ser jogado maximizando a recompensa total r_t, mostrada na equação 14.

$$\sum_{t=0}^{n} r_t = \sum_{t}^{n-1} (score(s_{t+1}) - score(s_t)) = score(s_n) - score(s_0) \tag{14}$$

Para escolher as ações a serem tomadas, os autores usam uma política de gradiente. Essa política reforça positivamente as ações que recebem recompensas positivas e, negativamente as ações que recebem recompensas negativas. O modelo usado para implementar a política é uma rede neural convolutiva. Essa rede convolui a matriz de adjacência do MIG e usa a função de ativação ReLU.

O trabalho de [Haaswijk et al. 2017] é o primeiro a mostrar a aplicação de aprendizagem de máquina em síntese lógica. Os resultados obtidos foram para vários casos próximos ou até, para alguns casos, melhores do que os algoritmos do estado da arte em síntese lógica.

4. Método Proposto

Este trabalho propõe a aplicação de boolean matching usando um jogo de aprendizagem por reforço inspirado nos métodos de equivalência de funções booleanas baseados em representação canônica. As funções booleanas são representadas por inteiros que correspondem aos estados s. As ações a do jogo correspondem às transformações permitidas

segundo o tipo de matching (P, NP, PN ou NPN) de acordo com a tabela 3. Assim como [Haaswijk et al. 2017], escolher a ação a em um estado s_t corresponde a transformação a função booleana de modo que uma nova função seja obtida no estado s_{t+1}.

O objetivo do jogo é encontrar uma função booleana que represente a classe de equivalência da função dada como entrada. Para isso, define-se uma função $score(s)$ que avalia o estado s segundo o quão próxima a função atual está da função representante da classe de equivalência. A recompensa r é dada por $r(s_t, a_t) = score(s_t) - score(s_{t-1})$. O jogo é jogado maximizando a recompensa total r_t, definida na equação 14.

A função escolhida para ser representante é a menor função da classe de equivalência. Ou seja, o objetivo do jogo é chegar ao menor $score$ possível. Para isso, uma política de gradiente inverso é usada. Assim, recompensas negativas recebem reforços positivos, e recompensas positivas recebem reforços negativos.

A implementação da política é feita através de redes neurais. Para cada ação a, uma rede neural que toma como entrada uma função booleana e os comportamentos de suas variáveis segundo a tabela 2 é construída. As redes dão como saída a probabilidade de sua ação a correspondente ser a ação ótima do jogo para transformar a função de entrada na menor função booleana da classe de equivalência.

O boolean matching é realizado aplicando separadamente o jogo de aprendizado por reforço a duas funções booleanas f_1 e f_2. Se $f_{1-menor} \equiv f_{2-menor}$, então f_1 e f_2 pertencem à mesma classe de equivalência. As transformações para ir de f_1 a f_2 são obtidas da sequência de ações tomadas no jogo de f_1.

5. Cronograma

O cronograma de atividades previstas para a realização do TCG2 é mostrado na tabela 7.

Tabela 7. Cronograma de Atividades

	Fev	**Mar**	**Abr**	**Mai**	**Jun**	**Jul**
Modelagem da Implementação	x	x				
Implementação		x	x	x		
Avaliação de Desempenho do Modelo de Aprendizagem por Reforço			x	x		
Análise dos Resultados				x	x	
Apresentação						x

A **modelagem da implementação** prevê a criação dos modelos UML e o processo de engenharia de software do código a ser implementado. A **implementação** prevê a escrita do código. A **avaliação de desempenho do modelo de aprendizagem por reforço** prevê a aplicação dos métodos de avaliação de modelo de aprendizagem de máquina para validar a performance do modelo treinado. A **análise dos resultados** prevê a crítica sobre os resultados obtidos.

Referências

Abdollahi, A. (2008). Signature based boolean matching in the presence of don't cares. In *2008 45th ACM/IEEE Design Automation Conference*, pages 642–647.

Abdollahi, A. and Pedram, M. (2008). Symmetry detection and boolean matching utilizing a signature-based canonical form of boolean functions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(6):1128–1137.

Assodiky, H., Syarif, I., and Badriyah, T. (2017). Deep learning algorithm for arrhythmia detection. In *2017 International Electronics Symposium on Knowledge Creation and Intelligent Computing (IES-KCIC)*, pages 26–32.

Bailey, B. (2017). Eda challenges machine learning. *Semiconductor Engineering*.

Boole, G. (1853). *Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities*. Also available from Dover, New York 1958, ISBN 0-486-60028-9.

Haaswijk, W., Collins, E., and Seguin, B. (2017). Deep learning for logic optimization. In *2017 International Workshop on Logic Synthesis (IWLS)*.

Kaggle (2017). Feature engineering. https://s3.amazonaws.com/dq-content/186/overfitting.svg. Acessado em 23/12/2017.

Kapoor, B. (1995). Improved technology mapping using a new approach to boolean matching. In *Proceedings the European Design and Test Conference. ED TC 1995*, pages 86–90.

Katebi, H. and Markov, I. L. (2010). Large-scale boolean matching. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 771–776.

Lau, M. M. and Lim, K. H. (2017). Investigation of activation functions in deep belief network. In *2017 2nd International Conference on Control and Robotics Engineering (ICCRE)*, pages 201–206.

Mailhot, F. and Micheli, G. D. (1990). Technology mapping using boolean matching and don't care sets. In *Proceedings of the European Design Automation Conference, 1990., EDAC.*, pages 212–216.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Nguyen, N. D., Nguyen, T., and Nahavandi, S. (2017). System design perspective for human-level agents using deep reinforcement learning: A survey. *IEEE Access*, 5:27091–27102.

Petkovska, A., Soeken, M., Micheli, G. D., Ienne, P., and Mishchenko, A. (2016). Fast hierarchical npn classification. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484 EP –. Article.

Swingler, K. (1996). *Applying Neural Networks: A Practical Guide*. Academic Press.

Wang, S. (1995). Artificial neuron and method of using same. US Patent 5,390,136.

Zhang, J., Yang, G., Hung, W. N. N., and Wu, J. (2017a). A canonical-based npn boolean matching algorithm utilizing boolean difference and cofactor signature. *IEEE Access*, 5:27777–27785.

Zhang, J., Yang, G., Hung, W. N. N., and Zhang, Y. (2017b). An efficient NPN boolean matching algorithm based on structural signature and shannon expansion. *CoRR*, abs/1708.04597.